

Cache, Cache Everywhere, Flushing All Hits Down The Sink: On Exclusivity in Multilevel, Hybrid Caches

Raja Appuswamy
Vrije Universiteit, Netherlands
raja@cs.vu.nl

David C. van Moolenbroek
Vrije Universiteit, Netherlands
dcmoole@cs.vu.nl

Andrew S. Tanenbaum
Vrije Universiteit, Netherlands
ast@cs.vu.nl

Abstract—Several multilevel storage systems have been designed over the past few years that utilize RAM and flash-based SSDs in concert to cache data resident in HDD-based primary storage. The low cost/GB and non-volatility of SSDs relative to RAM have encouraged storage system designers to adopt inclusivity (between RAM and SSD) in the caching hierarchy. However, in light of recent changes in hardware landscape, we believe that in the future, multilevel caches are invariably going to be hybrid caches where 1) all/most levels are physically collocated 2) the levels differ substantially only with respect to performance and not storage density, and 3) all levels are persistent. In this paper, we will investigate the design tradeoffs involved in building exclusive, persistent, direct-attached, multilevel storage caches. In doing so, we will first present a comparative evaluation of various techniques that have been proposed to achieve exclusivity in distributed storage caches in the context of a direct-attached, hybrid cache, and show the potential performance benefits of maintaining exclusivity. We will then investigate extensions to these demand-based, read-only data caching algorithms in order to address two issues specific to direct-attached hybrid caches, namely, handling writes and managing SSD lifetime.

I. INTRODUCTION

Over the last few years, advancements in fabrication processes and Flash Translation Layer (FTL) firmware have led to the evolution of flash-based solid state storage devices from low endurance storage media with random write issues to fast, energy efficient, reliable enterprise storage media. Despite low access latencies, the high cost/GB of flash with respect to HDDs has led to the adoption of flash as an intermediate caching tier rather than as a primary storage solution. Several research and commercial three-level (RAM-SSD-HDD) caching systems exist today and an aspect fundamental to the design of all these systems is that the RAM and SSD levels are managed independently using different placement (when and what data is admitted into the cache?) and replacement algorithms (how is data managed in the cache?). The disparity in cost/GB between RAM and flash combined with the volatility of RAM encouraged designers to opt for an inclusive hierarchy with SSDs redundantly caching data already cached in RAM.

Despite the successful adoption of these caching systems, we believe that recent changes in hardware landscape necessitate revisiting these design rationales. First, over the past few years, the rate of adoption of flash in enterprise storage has

been paralleled by an equally phenomenal drop in memory prices. This change has been embraced by both the hardware industry, as witnessed by the commoditization and consumer demand for manycore servers with large-memory DIMM form factors and high-bandwidth memory channels, and the software industry, as shown by the development of several low latency, RAM-only caching/primary storage systems like RAMCloud [13]. Today, several modern caching appliances use battery-backed, high-end enterprise servers, provisioned with several hundreds of GBs of RAM and SSD [1]. Second, recent advances in nonvolatile memory technologies have shown how Phase Change Memory (PCM)-based storage devices can blur the line between mass storage and memory. These devices promise high-density, non-volatile storage with latencies as low as a few microseconds (or even nanoseconds), with cost/GB much lower than RAM.

In light of the aforementioned changes, multilevel caches of today (and of the future) possess three salient properties. First, they are hybrid caches where multiple levels with different price/performance tradeoffs are physically collocated. Second, higher levels are not necessarily much smaller in size than the lower ones. Third, volatility is no longer an issue either due to the use of software/hardware techniques to deal with power failures (battery-backed logging and replay), or due to the persistent nature of new device families (PCM). These properties of modern multilevel caches invalidate the age-old assumptions made by designers for avoiding exclusion between levels.

In this paper, we explore the tradeoffs and design alternatives involved in building an exclusive, block-level, persistent, direct-attached RAM/SSD caching hierarchy. In doing so, our contributions are twofold. First, we show how maintaining exclusivity in a direct-attached, multilevel cache 1) improves performance by increasing the number of second-level hits, 2) potentially reduces operating expenses by enabling the usage of high-latency SSDs while still achieving performance comparable to an inclusive cache, and 3) increases SSD lifetime by significantly reducing the number of allocations in contrast to an inclusive cache. Second, we advance state of art by exploring the tradeoffs involved in building persistent, multilevel read-write caches. In particular, we will show how contrary to conventional wisdom, unifying the management of

both clean and dirty data under a cost-aware caching algorithm provides better performance than a partitioning the cache into autonomously-managed read/write regions.

II. EXCLUSIVITY IN DISTRIBUTED CACHES

It is well known that in a multilevel caching hierarchy, when cache levels are of comparable size, the redundancy of data between levels is extreme under inclusive caching [16]. In the most extreme case in a two-level RAM/SSD cache, for instance, both the RAM and SSD levels cache the same data, effectively halving the utility of the cache. Previously, researchers have investigated a similar problem in the context of distributed, multilevel caches. For instance, in enterprises, it is common for end-user requests to be serviced by front-end application servers (such as web servers), which in turn retrieve the data from networked, second-level storage servers. In such a scenario, both servers are likely to be equipped with large caches, which form a distributed, two-level hierarchy. The techniques that guarantee exclusivity in such hierarchies can be classified into two primary types depending on how the two cache levels are managed, namely, the *Global* technique and the *Unified* technique.

Under the Global technique, the application and storage server caches are managed independently similar to the inclusive case. However, in contrast to the inclusive case, where each read miss results in the same data being allocated at both first and second levels, the Global technique allocates data only in the first level. Allocation in the second level happens during eviction from the first level. In addition, to maintain exclusivity, any first-level read miss that finds the data cached at the second level results in the data being deleted from the second level (as it is allocated at the first level).

Several approaches have been proposed to implement the Global scheme in multilevel caches. The first approach (referred to as *Hint-based aggressive caching* [3]) involves adding an explicit *Demotion* operation between the two cache levels. The first level invokes this operation every time a block is evicted to allocate the target block in the second level. Although this approach offers a higher hit ratio, its performance is directly affected by the network bandwidth between the two servers. When bandwidth is scarce, demotions become expensive and overall performance deteriorates. Similarly, when the workload has little locality, this approach wastes bandwidth by unnecessarily demoting all data.

To solve these problems, the second approach (also referred to as *Hierarchy-aware reload* [3]) involves passively monitoring the first level data access patterns and using it to automatically infer evictions at the second level. For instance, in [4], the authors suggest maintaining a client content tracking table (CCT) that maintains information that maps memory regions in the application server's cache to data blocks. Thus, when a read request is received to transfer a block to a memory region which previously contained some other data, an eviction can be inferred. This information is then used to refetch recently evicted data into the second level cache in the background. Although this approach avoids the

network bandwidth issues between the two cache levels, it has been shown that background reads significantly increase the average miss penalty and nullify the benefits of exclusive caching [5].

The Unified technique (also referred to as *Client-controlled caching*), on the other hand, obviates demote operations by managing the two caches as a single unit. For instance, in ULC [7], a single instance of the cache management algorithm runs in the first level and keeps track of contents of both the application and storage server caches. When a block is accessed, ULC attempts to place a block at the appropriate level depending on its predicted re-reference distance (the number of requests serviced between two successive references to a block). Karma [18] is another example of the Unified technique where per-block access statistics are used to rank and assign blocks to levels depending on their importance. Despite being intuitively simple, recent research has shown that the unified technique suffers from two major issues [3]. First, it requires making extensive changes to storage client/server software and I/O interfaces. Second, it complicates the scenario where multiple application servers share a single storage server.

III. CASE FOR EXCLUSIVITY IN HYBRID CACHING

Achieving exclusivity in direct-attached, multilevel cache requires making different sets of trade offs in contrast to the distributed case. First, both caches are collocated in a single host with no network in between. This effectively eliminates the bottlenecks associated with both Global and Unified techniques. The Global technique benefits due to cheap demotions and the Unified technique benefits due to the ease with which data management across cache levels can be unified. Thus, it is important to compare these distributed techniques in the local context to see if the exclusivity offered by techniques provides any performance benefit under enterprise storage workloads.

Second, although demotions are cheap, they are not free, as each demotion equates to a slow flash write operation. Thus, it is important to understand if exclusive caching offers any improvement despite the flash write bottleneck, and if those performance benefits are sensitive to variations in flash write latency.

Third, the second level in our hybrid cache is flash-based unlike distributed caches where it is RAM-based. Although modern SSDs have high endurance and can sustain a large number of erasures, the design for any hybrid caching algorithm should consider SSD lifetime as a serious factor.

Fourth, unlike all the aforementioned studies, we are considering scenarios where both our caches are persistent. While all prior research ignores caching dirty writes, achieving optimal performance in our case requires investigating algorithms for a multilevel read-write cache.

Thus, to make a case for exclusivity in the local context, we need to answer the following questions:

- Does exclusive caching offer any performance improvement over the traditional inclusive caching approach?

- How sensitive is the performance of exclusive caching with respect to flash write latency?
- How sensitive is the performance of exclusive caching with respect to the RAM/SSD size ratio?
- If exclusivity is useful, which technique (Unified or Global) performs best?

In this section, using trace-driven simulation, we will make a case for exclusive caching in hybrid caches and answer all the aforementioned questions. Since research on achieving exclusivity in a distributed setting has primarily focused on read-only caching, we will present the results for a simulated, two-level, read-only hybrid cache. We will consider the problem of designing an integrated read-write caching algorithm in Section IV and managing SSD lifetime in Section V.

A. Simulation Environment

Opting for exclusion or inclusion is one of the two important design parameters in any caching system, the other parameter being the replacement algorithm used. In order to perform a fair comparison of the three cache management techniques (Inclusive, Global and Unified), we implemented a trace-driven simulator that extended the traditional Adaptive Replacement Cache (ARC) algorithm using these techniques. We chose ARC, as prior research has proved ARC to be the most effective, application-independent caching algorithm, and as it has been used in several distributed, multilevel caching studies. We will just summarize the ARC algorithm here due to lack of space and refer the reader to [10] for further details.

1) *ARC*: The fundamental idea behind ARC is to dynamically adapt to the recency or frequency changes in workloads by using two lists that separate recently seen pages from frequently accessed pages. The LRU list T1 contains pages that have been seen once while list T2 contains pages that have been seen more than once. Every hit in the cache results in the target page being moved to the top of the T2 list, while a miss in the cache adds a new page to the top of the T1 list.

In addition to these two lists, whose size is always enforced to be utmost “C” pages (where C is the cache size), ARC also maintains two LRU history lists B1 and B2. These lists store only page addresses (or block offsets) of “C” recently evicted pages without the actual data, and are populated each time a page is evicted from the T1 list (B1) or the T2 list (B2). Hence, these lists serve as ghost buffers that indicate potential hits that could have been achieved had the data been cached. Thus, although ARC manages a data cache of size C, it maintains metadata for 2C pages.

ARC uses these ghost buffers to dynamically adjust a parameter P that dictates the size of T1 (P) and T2 (C - P) lists. When a read miss finds a ghost page in the B1 list, ARC increases P, thereby allocating more pages to the T1 list. On the other hand if the ghost page is found in the B2 list, ARC decreases P, thereby allocating more pages to the T2 list. The lists themselves are truncated/expanded gradually by using P to guide in the selection of victim lists during page replacement. For instance, if $|T1| > P$, a page is evicted from T1, if not a page is evicted from T2.

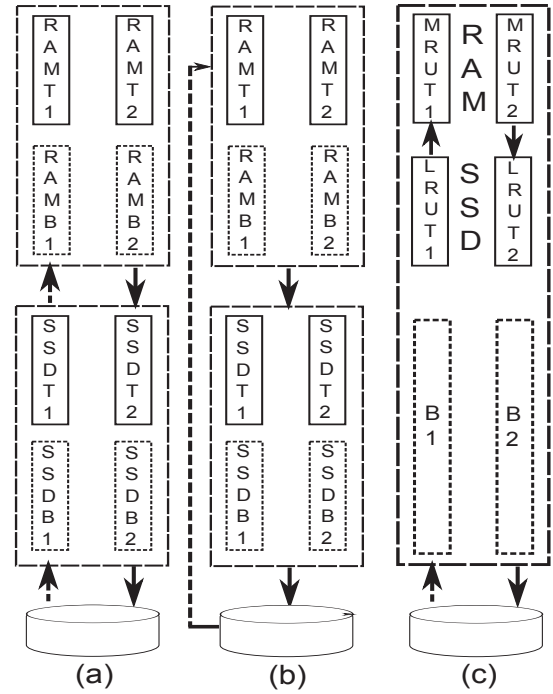


Figure 1: Comparison of the three different caching techniques ((a) Inclusive, (b) Global, and (c) Exclusive) used to manage a two-level RAM/SSD cache. Dashed rectangles in the figures represent ARC instances (two in Inclusive and Global case, and one in the Unified case). Solid rectangles represent T lists that track data pages and dotted rectangles represent B lists that act as ghost buffers. Dashed lines represent read misses and solid lines represent dirty data flushes. The figure also shows how ARC lists are managed under each technique.

2) *Two-level ARC Implementations*: Figure 1 depicts how ARC has been extended to handle our two-level hybrid cache. *Inclusive-ARC* (IARC) is the most straight forward case (Figure 1a). Here, our simulator runs two separate instances of ARC, one per level. Each instance maintains its own T-lists and B-lists and performs cache replacement independent of the other. Both ARC levels cache data on each cache miss. For instance, when a request to RAM ARC results in a miss, it is forwarded to the SSD ARC. If the data requested is cached in SSD ARC, it is propagated to RAM ARC which allocates it in its cache, replacing a block if the cache is full. If the data is not found in SSD ARC, then the block is allocated in both levels.

The *Global ARC* (GARC) implementation is similar to the inclusive ARC implementation in that the two levels still perform replacement independently. However, unlike inclusive ARC, Global ARC allocates data in the second level only during eviction from the first level as shown in Figure 1b. For instance, consider a read miss at the RAM ARC. In order to accommodate the new block, both inclusive and exclusive ARC variants pick a target. If the target is dirty (it has been written to), both variants write the block to the SSD ARC. But if the target is clean, only GARC writes it to the SSD ARC while IARC silently drops it. Similarly, on a read miss in the SSD ARC, while IARC allocates a block in both RAM and SSD ARCs, GARC allocates a block only in RAM ARC.

Device	Read Latency	Write Latency
SSD	25 μ s	200 μ s
HDD	5 ms	5 ms

Table I: Read/write performance characteristics of various devices used to compute virtual execution time

The *Unified ARC* (UARC) implementation considers the two levels as a single large cache as shown in Figure 1b. Thus, unlike IARC and GARC, our Unified implementation divides T1 and T2 lists among RAM and SSD while maintaining a single instance of B1 and B2 history lists. Thus, while RAM contains the top halves of the T1 and T2 while lists, SSD contains the bottom halves. It is important to note here that the lists might or might not have a bottom half in the SSD. The latter case would happen if ARC, as a part of its adaptation, has truncated the list to a size that fits entirely in RAM.

There are two major differences between UARC and GARC. First, while UARC maintains pages in the right lists across levels, GARC loses this information when a page crosses over to the SSD. Thus, under GARC, a page evicted from the T2 RAM list will be added to the T1 SSD list. Second, if R is the RAM size and S the flash size, GARC has a RAM ARC implementation managing $2 \times R$ (R cache + R history) pages and a SSD ARC implementation managing $2 \times S$ pages. UARC, on the other hand, has a single ARC implementation that manages $2 \times (R + S)$ pages. Thus, the rate of adaptation of parameter P , and hence the sizing of T1/T2 lists is different in the two implementations. For instance, if a read miss in GARC finds the page in RAM ARC’s B1 list, it increments parameter P even if the page is cached in the SSD - a second level hit is still a first level miss. Under UARC, on the other hand, both first and second level hits are considered alike. Hence, UARC would not increment parameter P . As we will show later, both these aspects cause noticeable performance differences between the two implementations.

3) *Simulator:* We implemented IARC, GARC and UARC in our simulation framework which allows us to benchmark these algorithms using a wide variety of real-life enterprise storage traces. The framework is implemented in C and has three major components, namely the parser, the hash manager, and the cache manager. The parser is a pluggable component that is trace specific and is responsible for translating block requests into a canonical form. The hash manager is a common set of routines used by various caching algorithms to keep track of cached blocks for quick search/insertion. The cache manager is another pluggable component that implements each variant of the aforementioned three algorithms.

The framework is instantiated using the RAM/SSD sizes, trace format, and cache management type as input. The framework passes each request generated by the parser to the cache manager one at a time and keeps track of the following statistics (among others): 1) number of hits in RAM, 2) number of hits in the SSD, 3) number of read misses that would need to be serviced by the HDD, 4) number of flushes (dirty data writes) to the HDD, and 5) number of demotions

Trace	I/Os (Millions)	Read (%)	Write (%)	R-WSS (GB)	RW-WSS (GB)
Financial1	36	15	85	1.11	3.66
Financial2	18	78	22	0.82	1.17
proj0	40	6	94	1.76	3.16
src10	406	47	53	120.76	121.26
src22	17	36	64	20.31	20.31
ts0	4	26	74	0.5	0.91
wdev0	3	27	73	0.2	0.53
web3	0.5	60	40	0.22	0.59

Table II: The table lists the following characteristics of each trace: the number of I/Os issued, read/write I/O percentages, and working set sizes which indicate the total amount of data accessed during the trace. R-WSS column indicates the working set of each trace when write requests are filtered out, while RW-WSS column includes write requests.

to the SSD. Using these metrics, the framework computes the virtual execution time for each trace as follows

$$VirtualTime = |SSD\ read\ hits| \times T_{SSDR} + |SSD\ demotions| \times T_{SSDW} + |read\ misses| \times T_{HDD} + |flushes| \times T_{HDD}$$

Since the total time taken by RAM hits is negligible in comparison to the overall execution time, we do not account for it in our cost model. Unless otherwise mentioned, the defaults we used for various device access latencies are listed in Table I. These values are identical to ones used in prior research [12] and model the behavior of a Maxtor Atlas 10K IV HDD and a SLC NAND flash memory.

B. Traces

We used two sets of traces that have been widely used in prior research for evaluating caching algorithms. The first set consists of two SPC traces (Financial1 and Financial2) that were collected by monitoring I/O requests of OLTP applications running at two large financial institutions [15]. The second set consists of a collection of traces from 36 different volumes across 13 servers running Windows Server 2003 SP2 [11]. Of these traces, we only used six traces that had at least 100,000 reads and writes, as our focus is on investigating the design tradeoffs involved in building a read-write cache. Table II summarizes the characteristics of these traces.

Having described the simulation framework and traces, we will now answer the questions we posed at the beginning of this section. Although the traces contain both read and write requests, as we mentioned earlier, the objective of this section is to compare various distributed exclusion techniques in the context of a direct-attached hybrid cache. So the results presented in this section are obtained by filtering out writes and using our caching implementations as read-only caches. Thus, in the following experiments, the cache size used is a fraction of the working set size reported in the R-WSS column of Table II.

C. Is Exclusivity Beneficial?

Figure 2 shows the hit rate achieved by our UARC implementation normalized by the IARC hit rate under various

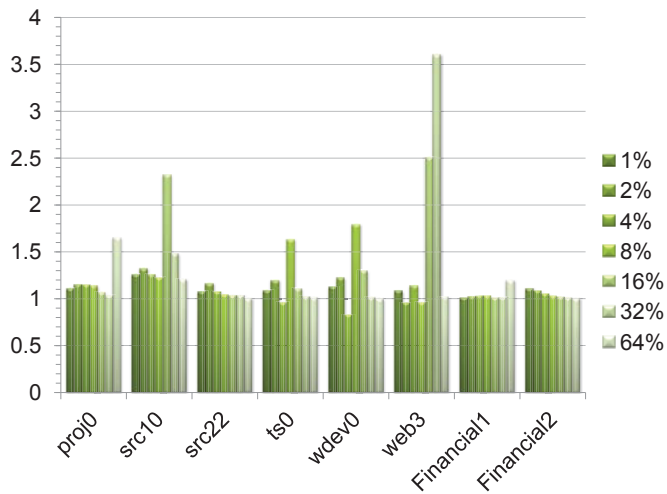


Figure 2: Total hit count achieved by the UARC normalized by IARC's hit count for each trace (x-axis)/cache size (fraction of working set size listed as legend entries) combination

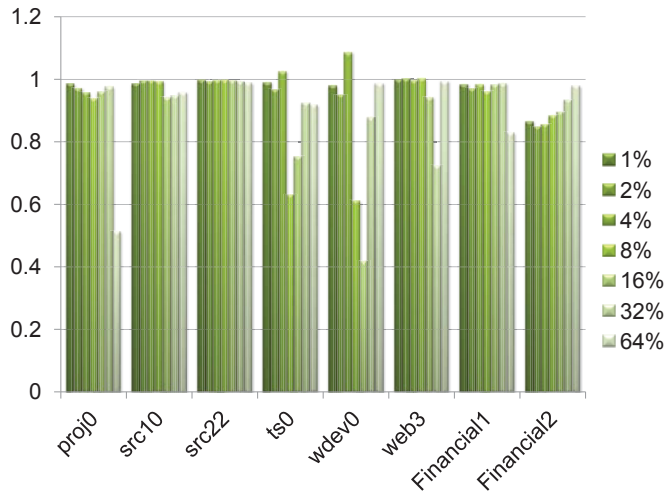


Figure 3: Execution time of UARC normalized with respect to IARC

traces. For each trace, the figure shows the normalized performance at each cache size ranging from 1% to 64% of the working set size (total amount of unique data accessed across the entire trace as shown in Table II). For this experiment, we assumed an equisized RAM/SSD cache. We will investigate the impact of varying the RAM/SSD size ratio later in this section.

There are two important observations to be made from the figure. First, under most trace-cache size combinations, UARC matches or improves the hit rates achieved by IARC. Figure 4 explains the reason behind this by showing a break down of the hits achieved by RAM ARC and SSD ARC under our Inclusive and Unified implementations with the Web3 trace. As can be seen in the figure, SSD ARC achieves a much higher hit rate under UARC due to exclusive caching, thereby boosting overall cache efficiency.

Second, at certain cache sizes UARC outperforms IARC by a significant margin than other sizes. For instance, under web3, at 32% cache size, UARC improves hit count by a factor of

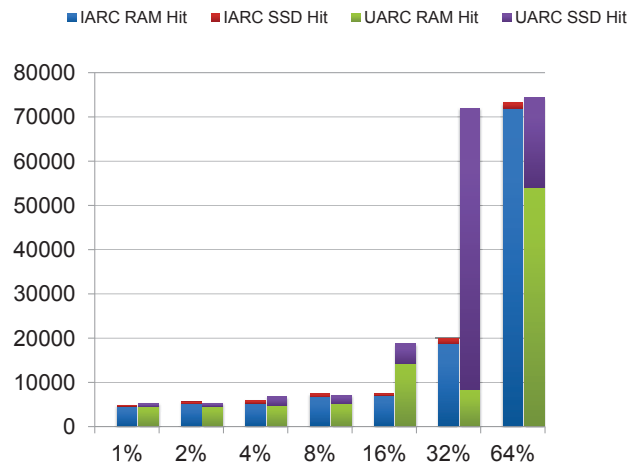


Figure 4: Total hit count achieved at the two levels (RAM and SSD) by both UARC and IARC under the trace Web3 at various cache sizes

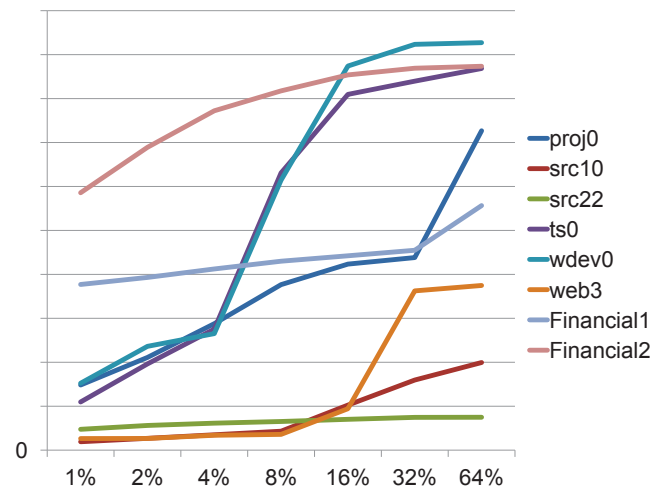


Figure 5: Hit rate (hits/total I/Os) achieved by UARC

3.5, while at 64%, IARC catches up. Looking at Figure 4, one can see that at 32% cache size, UARC achieves a significant increase in the number of second-level hits, while at 64%, the cache is large enough to satisfy most rereferences from the first level. This indicates that UARC can capture block rereferences at much smaller cache sizes than IARC due to its exclusive nature.

Figure 3 shows normalized UARC execution time. Comparing this with Figure 2, the treachery of using hit counts as the only performance metric is evident, as evidenced by prior studies [5]. While under certain traces, like Src1, UARC outperforms IARC impressively with respect to the total number of cache hits, it fails to register a similar improvement in overall execution time. Figure 5 shows that at 16% cache size, only 10% of Src1 requests are serviced by the cache. Thus, despite a significant increase in the hit rate, the overall execution times under both UARC and IARC are dictated by expensive cache misses.

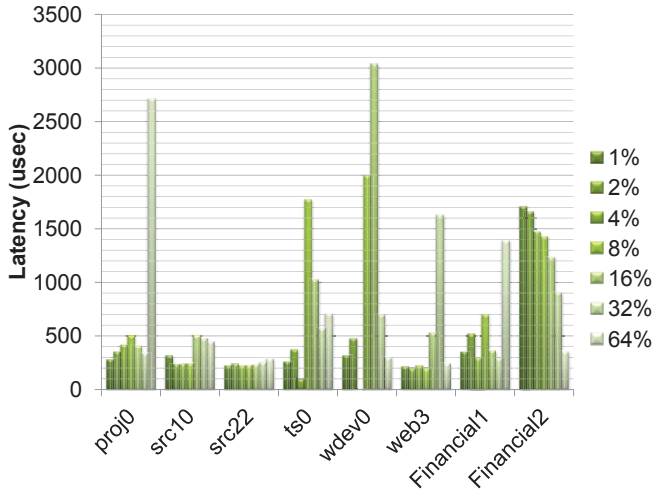


Figure 6: Sensitivity to SSD write latency: Figure shows the threshold SSD write latency at which UARC’s execution time degrades to that of IARC at 200 μ s. A latency higher than 200 μ s means UARC is able to match IARC’s performance despite using a slower SSD

UARC produces a 17.8% deterioration in hit rate and 8.2% increase in execution time only under Wdev0 at cache size 4%. On further investigation, we found this trace-cache size combination to be a pathological case for ARC itself. To confirm this, we wrote a separate single-level ARC implementation (henceforth referred to as SARC) without any of our two-level extensions. As expected, the number of hits achieved under UARC and SARC were identical in all cases. Similar to UARC, SARC also exhibited the 17.8% reduction in hit rate.

D. Sensitivity to Flash Writes

As we mentioned earlier, demotions in a direct-attached hybrid cache are cheap compared to the distributed setting, but not free as each demotion is a flash write operation. In order to understand the sensitivity of UARC’s performance to flash write latency, we recomputed UARC’s execution time for write latencies ranging from 200 μ s to 1ms. Using these values, we computed the threshold write latency at which UARC loses its performance edge over the default IARC configuration (which is based on a 200 μ s SSD).

Figure 6 shows the write latency threshold. As expected, greater the performance differential between UARC and IARC (in Figure 3), higher the latency threshold. This presents a price/performance tradeoff that can be used during provisioning a storage system to reduce overall capital expenses as one could either use UARC to manage low-latency, high-cost, SSDs to improve performance or reduce price by using cheaper, consumer-grade SSDs with higher write latencies while matching the performance offered by IARC.

E. Sensitivity to RAM-SSD Sizes

So far in our evaluation, we have restricted ourselves to configurations where RAM and SSD are equal in size. However, in order to understand how the RAM-SSD size ratio might have an impact on the effectiveness of exclusivity, we

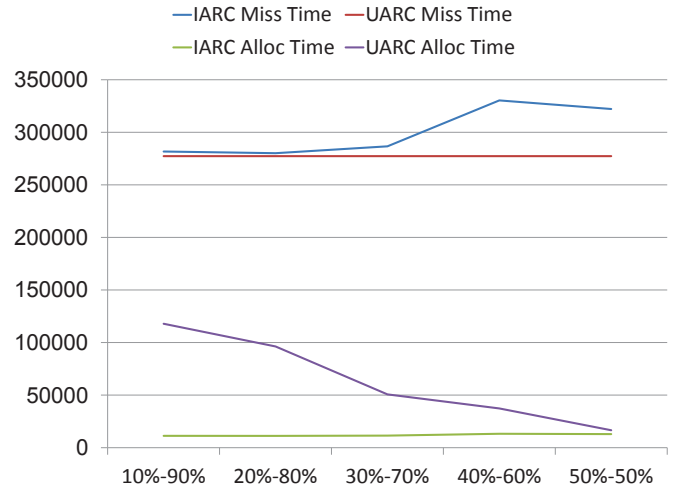


Figure 7: Variations in UARC/IARC miss/allocation times under wdev0 at 32% cache size while varying RAM/SSD size ratio

reran the simulations for each cache size while varying the ratio from 50%-50% RAM-SSD to 10% RAM-90% SSD.

Table III shows the threshold fraction of RAM below which UARC deteriorates performance (by at least 1%) when compared to IARC. For instance, under Wdev0 trace, at a cache size equal to 64% of the working set size, UARC outperforms IARC when the RAM size is 50%, 40%, and 30%. However, at 20% and below, UARC deteriorates performance compared to IARC. Thus, the table lists 20% as the threshold RAM size fraction and reports a deterioration of 29% in execution time.

As can be seen, under several traces-cache size configurations, UARC is able to match or outperform IARC at all RAM-SSD ratios as shown by the “-” entries in the table. Thus, even traditional systems with low RAM/SSD ratios would benefit from exclusive caching. Even when UARC deteriorates performance, it does so only by a small fraction (less than 5% in all cases except under wdev0 and ts0).

The drop in UARC performance under wdev0 and ts0 at cache size of 4% is consistent with Figure 3, and is due to that trace-cache size combination being a pathological case for ARC as we mentioned in Section III-C. The performance drop at cache sizes of 32% and 64%, on the other hand, is explained by Figure 7, which shows the variations in miss time ($|\text{cache misses}| \times T_{hd}$) and allocation time ($|\text{demotions}| * T_{ssdw}$) of UARC/IARC under wdev0, at a cache size of 32%, while varying the RAM/SSD size ratio.

There are three interesting observations. First, UARC miss time remains a constant due to the fact that irrespective of the RAM/SSD ratio, the total cache size, and hence, the total number of hits in the cache, remains fixed.

Second, IARC produces fewer hits as the relative RAM size increases, as evidenced by the increase in miss time. This clearly highlights the pitfall of using inclusion in the cache hierarchy. As we mentioned earlier, under IARC, RAM and SSD caches are managed independently and might end up inclusively caching the same data twice. Thus, irrespective

Trace	1%	2%	4%	8%	16%	32%	64%
Financial1	-	-	-	-	-	-	-
Financial2	-	-	10% (1.031)	10% (1.0216)	10% (1.0404)	20% (1.0543)	20% (1.0311)
hm0	-	20% (1.0132)	-	-	-	-	-
proj0	10% (1.0144)	10% (1.0113)	-	-	-	-	30% (1.0378)
src10	-	-	-	-	-	-	-
src22	-	-	-	-	-	-	-
ts0	10% (1.0102)	10% (1.0198)	50% (1.0471)	-	30% (1.1393)	20% (1.1460)	10% (1.1102)
wdev0	-	-	50% (1.1182)	-	-	30% (1.3429)	20% (1.2995)
web3	-	-	-	-	-	10% (1.0111)	40% (1.0189)

Table III: RAM/SSD size sensitivity: Table shows the threshold RAM size (fraction of the overall cache size listed as column headers) at which UARC performs worse than IARC. The value within brackets is normalized UARC execution time (UARC time/IARC time). Entries with a value “-” denote cases where UARC matches/outperforms IARC at all RAM-SSD ratios (10%-90% to 50%-50%)

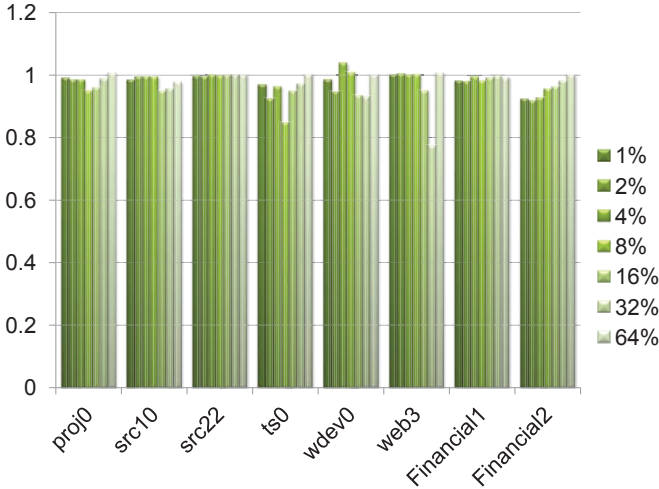


Figure 8: Normalized UARC execution time (with respect to GARC)

of the cache size, the effectiveness of an inclusive multilevel cache is essentially determined by the larger of the two caches. Thus, at low RAM sizes, the increase in the number of SSD ARC hits far outweighs the decrease in RAM ARC hits. However, at higher RAM sizes, this is no longer the case, and thus the effective hit rate degenerates to the hit rate of just one level, thereby increasing the miss time.

Third, UARC allocates almost an order of magnitude more blocks on the SSD than IARC at low RAM/SSD ratios. This can be explained by the difference in placement logic between IARC and UARC. While IARC allocates a block in the SSD during a read miss, UARC allocates it when it is evicted from RAM. Hence, decreasing the RAM size has the effect of increasing the number of demotions, and hence, the number of allocations. If this demotion-induced increase in execution time due is not offset by a proportionate reduction in miss time (due to the increase in SSD ARC hits), which happens in the case of wdev0 and ts0 at low RAM/SSD ratios, then exclusivity degrades performance. This explains why IARC outperforms UARC at low RAM/SSD ratios.

F. Unified vs Global

Figure 8 presents the execution time of our UARC implementation normalized with respect to GARC’s execution time. It is evident that UARC meets or outperforms GARC at all

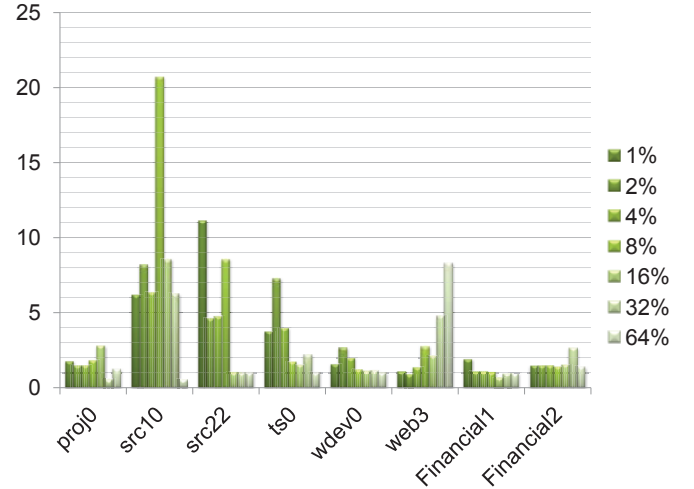


Figure 9: Total hit count achieved by SSD ARC under UARC normalized with respect to GARC

configurations except the pathological case (wdev0). Now, we will show how the two differences between GARC and UARC play a role in creating this performance differential.

Figure 9 shows the number of hits achieved by SSD ARC using UARC normalized by the corresponding GARC values. Clearly, UARC provides a much higher hit rate than GARC under most configurations. Although GARC maintains exclusivity between caches, it lags behind UARC as it fails to pass semantic information between layers as we mentioned earlier in Section III-A2. All blocks evicted from RAM ARC (including those from T2 list) are added to the T1 list of SSD ARC under GARC in contrast to the UARC which maintains block-list associations across the RAM-SSD barrier. This difference accounts for the differential in SSD ARC hit rates.

Figure 10 shows the RAM ARC hits achieved by UARC normalized by corresponding GARC values. It is interesting to see how under several traces, UARC achieves fewer hits than GARC. As GARC maintains B1 and B2 lists separately for RAM ARC and SSD ARC, its rate of adaptation of parameter P is different compared to UARC. This results in more RAM ARC hits under GARC than UARC. However, it is important to note here that the improvements in hit rates do not always

translate to overall improvement as can be seen by comparing Figures 8, 9, and 10.

G. Summary

Based on the results, maintaining exclusivity in a direct-attached, persistent, hybrid, multilevel caching hierarchy improves overall performance by boosting the SSD ARC hit rate while being resistant to changes in both SSD write latencies and RAM-SSD size ratio. As Unified caching provides non-negligible performance improvement over Global caching, for the rest of this paper, we will use UARC as our baseline multilevel implementation.

IV. TOWARDS A UNIFIED, MULTILEVEL, READ-WRITE CACHE

Having illustrated the benefit of exclusive caching under read-only workloads in the previous section, we now turn our attention to the topic of integrating writes into our multilevel cache implementations. Thus, all results reported from here on differ from previous ones with respect to two factors: 1) unlike before, the caches serve all requests in a trace without filtering out writes, and, 2) cache sizes reported in graphs are fractions of the working set size reported under column RW-WSS (not R-WSS) in Table II.

While a lot of work has been done on achieving exclusivity in read-only caches, to our knowledge, no work has focused on caching dirty data in an multilevel scenario. However, there has been some research on managing single-level read-write caches. Researchers have suggested two ways of integrating writes into a read-only cache. The first approach involves partitioning the cache into read/write regions and managing them separately [2], while the second approach involves modifying existing demand-based, read-cache algorithms, or implementing new ones, that are targeted at not just improving the hit rate but reducing expensive disk I/O [8] [9].

We will now investigate the applicability of these techniques in a multilevel scenario. But before doing so, we need to quantify the impact of exclusivity under a read-write workload to understand if UARC retains its performance edge over IARC when used as a read-write cache.

A. Exclusivity of Dirty Data

Achieving exclusivity in the presence of writes requires two things. First, just like clean data, dirty data should always be allocated in RAM ARC and be demoted to SSD ARC during eviction. Second, when an existing data block is overwritten, the old version, which could be in SSD or RAM ARC, should be invalidated. While both IARC and UARC satisfy the first condition, only UARC is capable of supporting the latter case.

Figure 11 shows the execution time of UARC normalized by IARC under all traces for both read and write I/Os. Comparing this with Figure 3 (the normalized execution time under these traces with write requests filtered out), one can see that maintaining exclusivity amidst writes offers even higher performance benefits across all cache sizes. This improvement stems from a reduction in read misses and number of flushes,

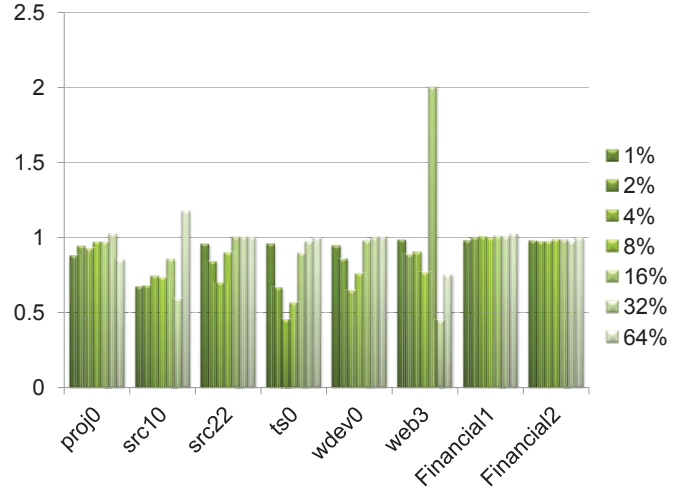


Figure 10: Total hit count achieved by RAM ARC under UARC normalized with respect to GARC

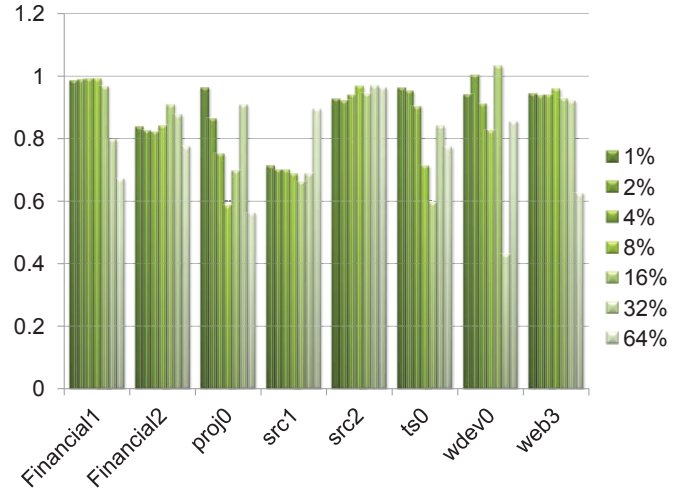


Figure 11: Figure shows the execution time of UARC normalized with respect to IARC. Unlike Figure 3, where UARC/IARC functioned as a read-only caches due to write requests being filtered out from traces, this figure reflects the behavior of UARC/IARC as read-write caches

both of which can be directly attributed to the exclusive nature of UARC. In order to consider how exclusivity reduces the number of flushes, consider an overwrite request for a block that is cached by SSD ARC. While UARC invalidates the old copy in SSD ARC, IARC leaves it unmodified. If this SSD block was clean, the performance deterioration is limited to reduction in SSD hit rate, as it will be dropped silently during eviction. However, if the block was dirty, in addition to occupying precious cache space, it also causes an expensive HDD flush when evicted. By maintaining exclusivity of both clean and dirty data, UARC eliminates these issues by design.

B. Partitioned Multilevel Cache

Prior research has shown how employing a dynamic partitioning technique that divides the cache into read-write regions, and managing each region independently could help improve overall performance in a single-level, read-write

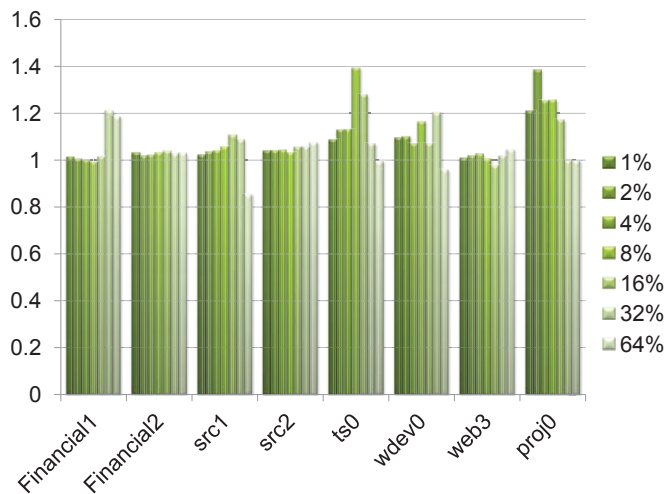


Figure 12: The best case execution of time P-UARC normalized with respect to baseline UARC’s execution time

cache [2]. Although no work has been done in extending this technique to a two-level cache, researchers have suggested transforming read-only/write-only caches into read-write variants by using a multilevel partitioning technique as the right way to manage a read-write cache [17].

The first design parameter involved in building a partitioned read-write cache is the replacement algorithm to be used for caching dirty data (we use ARC for clean data). Unlike most single-level write caching research that uses LRU as the replacement algorithm, our experimentation revealed that ARC offered a substantial reduction in write misses compared to LRU, despite being a demand paging algorithm. Thus, we implemented a two-level, partitioned read-write cache simulator (P-UARC) using ARC as the replacement algorithm for managing both read and write regions.

P-UARC has two independent UARC instances, one managing clean blocks and another managing dirty blocks, running side by side. The only case when a block migrates between the two instances is as a side effect of a write operation that finds a clean block cached in the read instance. Thus, while the write instance managing dirty data serves both read and write requests, the clean-block-only read instance is used only for servicing read requests. In addition, our P-UARC implementation maintains locality information across instances by always adding a block to the T2 list of the write instance during migration from the read instance.

P-UARC takes as input the RAM and SSD sizes just like UARC. In addition, it also takes a fraction that denotes the percentage of cache space to be allocated to each instance. Based on this fraction, it divides RAM and SSD space into read/write chunks and spawns two UARC instances to manage each chunk. Using P-UARC, we evaluated the impact of partitioning on overall performance by statically varying the read/write cache size ratio from 0.1 to 0.9. As there are a total of 560 different combinations (seven cache sizes, ten size ratios, and eight traces) we will just report the best case

execution time of P-UARC (that is, the lowest execution time across all size ratios).

Figure 12 shows the best case execution of P-UARC normalized by the corresponding execution time of baseline UARC we presented in Section IV-A. As can be seen, P-UARC either matches or lags behind UARC under most configurations. Further analysis showed us that while P-UARC definitely reduces the overall read miss rate, it does so at the expense of increasing the number of flushes. This effect gets compounded under traces where write hits caused by overwrites are equally (or more) likely than read hits caused by rereads. This clearly illustrates how partitioning offers no benefit when combined with adaptive algorithms like ARC in a multilevel, persistent cache.

C. Cost-Aware UARC

Recent research has shown how an algorithm that attempts to optimize per-block I/O costs outperforms one that focuses on optimizing only the overall hit rate [8] [9]. The fundamental idea behind such cost-aware algorithms is to pick the block with the lowest I/O cost during eviction, in contrast to a traditional read-caching algorithm that picks the block with the least recency/frequency.

In a single-level caching scenario, the I/O cost of evicting any block is the sum of the cost incurred during eviction and the cost incurred to fetch the block during next access. Thus, for a clean block, the eviction cost is zero as it can be dropped silently during eviction, while a dirty block has an eviction cost of one. Similarly, the cost of fetching a block during the next access is zero if the next access is a write operation, and one if it is a read. In [9], the authors show how a cost-aware replacement algorithm, which exploits next access type hints from a database query planner by grouping blocks into lists based on their I/O cost, is capable of outperforming state-of-the-art read-caching algorithms.

In this section, we will investigate the utility of cost-awareness by implementing a multilevel, cost-aware UARC algorithm (CA-UARC). Although we could implement an algorithm that relies on semantic hints from the file system or higher levels to predict the next access type, its applicability in real-life would be limited in comparison to a transparent, block-level caching algorithm that uses a workload-independent, generic prediction facility. In addition, the workload traces we use in this study are devoid of any file semantics. Thus, we will first describe two simple prediction strategies and the rationale behind their design before detailing our CA-UARC algorithm.

a) Dirty Bit Predictor: Our first predictor is targeted at traces in which blocks can be clearly classified as read-only and write-only types. In such cases, the type of next access to a block can be predicted with 100% accuracy without maintaining any history information as clean blocks are always read and dirty blocks are always overwritten.

b) Access Count Predictor: Our second predictor is based on the observation that even under traces that have blocks that are both read and written, it should be possible

Level	Status	NA	I/O Cost	List
RAM	Clean	R	2 (1 SSD W + 1 HDD/SSD R)	2
RAM	Clean	W	1 (1 SSD W)	1
RAM	Dirty	R	2 (1 SSD W + 1 HDD/SSD R)	2
RAM	Dirty	W	1 (1 SSD W)	1
SSD	Clean	R	1 (1 HDD R)	1
SSD	Clean	W	0	0
SSD	Dirty	R	2 (1 HDD W + 1 HDD R)	2
SSD	Dirty	W	1 (1 HDD W)	1

Table IV: I/O costs of each block in RAM and SSD caches

to classify blocks as read-mostly or write-mostly depending on the number of reads/writes. Thus, for each block, this predictor maintains read/write access counts, which it updates on each access. When queried for a prediction, it checks the target block’s read/write ratio. If the ratio is less than 0.5, it predicts a write and if it is greater than two it predicts a read. If these conditions are not met, it predicts the next access type to be the inverse of the previous access type based on the naive assumption that an access stream consisting of alternating read/write requests to blocks causes the read-write ratio to stay close to one.

c) *Ideal Predictor:* In addition to the aforementioned predictors, we also implemented an offline ideal predictor. The predictor is invoked before the simulation run and it builds an in-memory data structure that encodes all accesses in a compressed form. During the simulation run, the predictor keeps track of accesses on the fly and when queried, offers prediction with 100% accuracy. We implemented this predictor to both understand the best case performance benefits of adding cost-awareness to ARC, and quantify the performance loss due to mispredictions by other predictors.

1) *I/O Costs in a Multilevel Cache:* Since we are dealing with a multilevel caching scenario, the cost of blocks cached in RAM is different from those cached in the SSD. Table IV summarizes the total I/O cost for each block type (clean/dirty)-next access type (read/write) combination, for RAM and SSD cases separately. Given these I/O costs, it is evident that RAM-resident cached blocks can be classified into two types with costs one (write after read, write after write), or two (read after read, read after write), and SSD-resident ones into three types with costs zero (write after read), one (write after write, read after read), or two (read after write).

2) *CA-UARC:* In order to make UARC consider I/O costs in addition to locality as a replacement criteria, we extended it by first dividing each cache (RAM and SSD) into two segments, namely, a *time segment* of segment that is $\lambda\%$ of the cache size, and a *cost segment* that is $(1 - \lambda)\%$ of the cache size, where λ is a statically configurable parameter. We partition the T1 and T2 lists across these segments as shown in Figure 13. Thus, the RAM resident MRU-part of both T1 and T2 lists is internally decomposed into three sublists, namely, a time-LRU sublist (RAM-TL), a cost1-LRU sublist (RAM-C1L), and a cost2-LRU sublist (RAM-C2L). Similarly, the SSD-resident, LRU parts of the T1 and T2 lists are subdivided into time-LRU (SSD-TL), cost0-LRU (SSD-C0L), cost1-LRU (SSD-

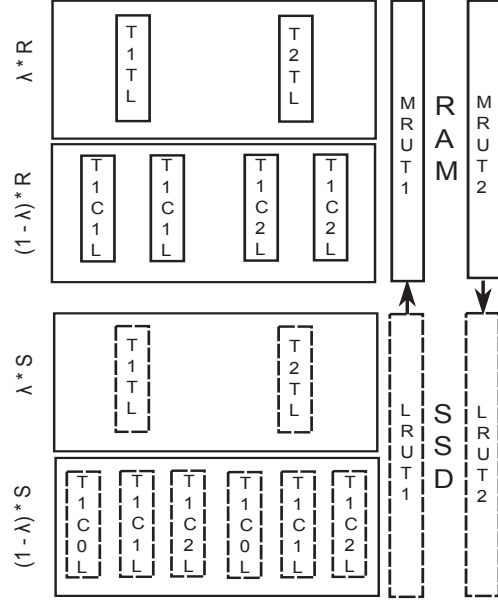


Figure 13: CA-UARC: Figure shows our extensions to the baseline UARC data structures for implementing cost awareness. In the figure, R and S denote RAM and SSD sizes, and λ is the fraction that determines the size of time and cost lists.

C1L), and cost2-LRU (SSD-C2L) sublists. Thus, by varying λ , we can change the behavior of CA-UARC from an being entirely ARC-based to entirely cost-based.

Similar to traditional ARC, each block access results in the target block being added to the RAM T1 or T2 list (depending on the access being a hit or a miss). More specifically, the block is added to the top of RAM-TL of either T1 or T2 list. If adding the block results in the time segment ($|T1\text{-RAM-TL}| + |T2\text{-RAM-TL}|$) exceeding the preconfigured size, the LRU block from the block’s new RAM-TL is removed. (For instance, on a block miss, the LRU block of T1 list’s RAM-TL would be removed). The predictor is then invoked to determine the next access type for the target block, and depending on the prediction, the block is added to the top of either RAM-C1 or RAM-C2. (For instance, assuming that the target block removed from the T1 RAM-TL sublist is clean, and assuming that the next access type is predicted to be a read, the block will be added to T1 RAM-C2).

If adding the block to the cost list results in the cost segment ($|T1\text{-RAM-C1L}| + |T2\text{-RAM-C1L}| + |T1\text{-RAM-C2L}| + |T2\text{-RAM-C2L}|$) exceeding the preconfigured size, a target block has to be chosen for demotion. This happens in two stages. First, the ARC logic is used to select the victim list (T1 or T2) following which, the LRU block from the victim’s RAM-C1L is chosen for demotion. If RAM-C1L is empty, RAM-C2L is victimized, and if both cost lists are empty, then the LRU RAM-TL is victimized. Thus, we always demote the least expensive (lowest I/O cost) block.

The demoted block is added to the appropriate SSD-TL (For instance, if the block being demoted was resident in the RAM

Size	Fin1			Fin2			proj0			src1			src2			ts0			wdev0			web3		
	I	DB	AC	I	DB	AC	I	DB	AC	I	DB	AC	I	DB	AC	I	DB	AC	I	DB	AC	I	DB	AC
16%	0.81	1.02	1.01	0.83	1.02	1.00	1.00	1.03	1.03	1.00	1.00	1.01	0.96	1.00	1.00	0.93	1.01	1.01	0.90	0.95	0.94	0.76	0.97	0.96
32%	0.79	1.04	1.04	1.07	1.02	0.99	0.93	1.01	1.01	0.96	1.02	0.98	0.92	1.00	1.00	0.85	1.02	1.02	0.95	1.02	1.01	0.73	0.94	0.90
64%	1.27	1.03	1.04	1.40	1.00	1.00	0.93	1.00	1.00	0.78	0.99	0.76	0.84	1.00	0.91	1.09	0.98	0.98	1.07	1.04	1.04	1.14	0.98	0.98

Table V: Table lists the best case CA-UARC execution time across all λ values normalized with respect to baseline UARC. For each trace (listed as column headers), we report the results achieved by CA-UARC while using the ideal predictor (I), dirty bit predictor (DB), and access count predictor (AC)

T1 list, it would be added to the T1 list’s SSD-TL). If adding this block results in an overflow in SSD-TL, the LRU block is removed and moved to the appropriate cost list similar to the RAM case. If this cost segment overflows, SSD-C0L, C1L, C2L, and TL are checked, in that order, to select an eviction target. If the target is dirty it is flushed back to the HDD, and if clean, it is silently discarded.

While there are other ways to implement cost-aware extensions to ARC, and we did experiment with a few suboptimal alternatives (like using the RAM cache as a time segment and dividing just the SSD into cost segments, or adding cost-awareness just to the RAM cache), our CA-UARC has the following advantages. First, by integrating cost-awareness within ARC (compared to an split ARC-cost-aware replacement combination), we preserve all salient ARC properties, like resistance to sequential scans and adaptivity to changing workloads. Second, by imparting cost-awareness to the RAM and SSD cases separately, we optimize both demotion costs and eviction costs.

Having described our CA-UARC implementation, we will now present our performance analysis using the cost-unaware UARC implementation as our baseline. As there are 1848 possible combinations (11 λ values from 0 to 0.9, three prediction types, seven cache sizes, and eight traces), we will just present summary statistics that report the best case performance of CA-UARC (across all λ values) for each trace-cache size-predictor type combination.

3) *Benefit of Being Cost Aware - The Ideal Case:* Table V shows CA-UARC’s best case execution time (across all λ values) normalized by the baseline UARC value. We do not report the results for cache sizes smaller than 16% as we saw no performance improvement at these sizes. The column labeled “I” under each trace shows CA-UARC’s execution time using the offline-optimal predictor, normalized by that of baseline UARC. While the reduction in execution time varies across configurations, the results clearly show that in the presence of an oracle, CA-UARC reduces the execution time by as much as 23%, proving the utility of I/O cost as a replacement metric.

An interesting observation from the table is how, despite being supported by an omniscient oracle, the CA-UARC implementation degrades performance at the highest cache size. On further investigation, we found this to be an unexpected side effect of one of our optimizations. As the ideal predictor knows all future references for a block, it prioritizes the eviction of blocks that are never going to be accessed again by queuing them on a special “end-of-reference” list. Our CA-UARC implementation evicts blocks from this list first before moving to the actual cost lists. At lower cache sizes, this

improves performance by caching only blocks with pending accesses. However, at large cache sizes, this has the effect amplifying disk flushes as the idealizer evicts dirty blocks that have no accesses before clean blocks in the cost list.

4) *Cost of Mispredictions - The Realistic Case:* Table V also shows the normalized execution time of CA-UARC while using DB and AC predictors. While these predictors do offer some improvement in performance under wdev0 and web3 workloads, they deteriorate performance even in the best case under other traces. On investigating this further, we found that both predictors reduce read misses significantly. But the reduction in read misses was at the expense of an increase in flushes similar to the P-UARC case. Thus, under traces where overwrites occur within very short intervals, the higher hit rate offered by the I/O cost-based replacement metric is easily overshadowed by the incessant flushing of dirty data.

Exception to this general behavior are the results under traces src1 and src2. Under these traces, the AC predictor offers significant reduction in execution time. To our surprise, we found that this was purely due to a dramatic reduction in the number of flushes unlike other traces. Further analysis showed us that most blocks were read/written an equal number of times under these traces, thereby reducing the AC predictor effectively to the guess case. As src1, src2 are source control server traces, they have alternating sequences of reads and writes, which the guess case predicts accurately. Thus, the AC predictor improved performance by being domain/trace specific.

5) *Summary:* Our evaluation shows how modifying replacement logic to balance clean and dirty data in a read-write cache is capable of improving performance in contrast to managing a partitioned cache with disparate caching algorithms. However, more work is required to understand if sufficient prediction accuracy can be achieved in the absence of higher-level hints or domain-specific semantic information. Although we have not investigated the effect of destaging on these algorithms, we believe that these advantages would only be amplified, as it is well known that any algorithm with a higher destaging rate will adversely affect performance due to interference of background writes with foreground reads [6].

V. DEALING WITH SSD LIFETIME

Endurance problems caused by limited erasure counts has been one of the major issues hindering the adoption of flash as the primary storage medium. However, recent research has shown that the skewed block popularity distribution inherent to several storage workloads can be used to allocate only “popular” blocks in the SSD, thereby improving both lifetime and overall cache effectiveness [14]. In this section, we

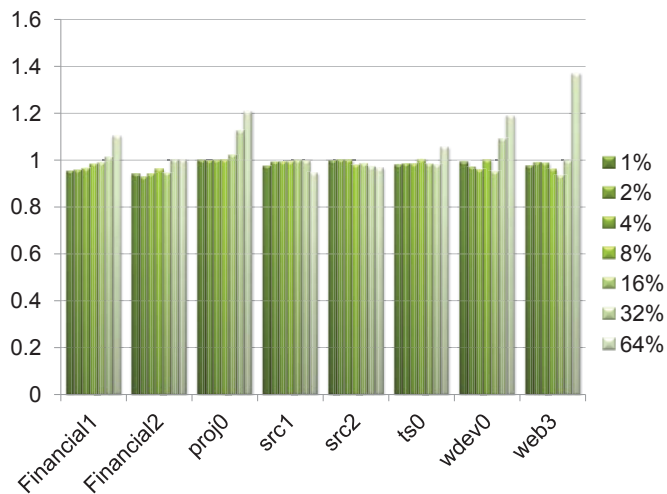


Figure 14: SE-UARC’s best-case execution time (across all sieve thresholds) normalized by that of baseline UARC

investigate the problem of sieving demotions to the SSD in the context of our multilevel cache in order to answer the following questions.

- Does sieving improve performance in a multilevel cache?
- Assuming we optimize lifetime over performance, what is the worst-case deterioration in execution time when demotion-based allocation is at its minimum?
- How does sieving with interact with the cost-aware replacement?
- How does UARC compare to IARC in the presence of sieving with respect to both performance and SSD lifetime?

A. Adding Sieving to UARC/IARC

Sieving uses a per-block popularity metric to distinguish “hot” blocks that are worth caching from “cold” ones that just thrash the cache [14]. We implemented sieving-enabled IARC (SE-IARC) and sieving-enabled UARC (SE-UARC) by maintaining per-block access counters in SSD ARC, which are incremented during each read/write/demote access from RAM ARC. Both implementations admit a block into the SSD ARC only if its access count is higher than the preconfigured sieving threshold. Blocks that fail this test are silently dropped if clean, and flushed out to the disk if dirty. We would like to explicitly point out here that SE-UARC does not perform cost-aware replacement like CA-UARC. We used SE-UARC to investigate the effect of sieving in isolation. We will describe the interaction between sieving and cost-aware replacement later in this section.

Having described our sieving-related extensions, we will now answer the questions we posed earlier in this section using our simulation results. To gather these statistics, we ran the simulator multiple times while statically varying the sieving threshold from one to 64 (powers of 2) block accesses. As before, we will limit ourselves to reporting just summary statistics.

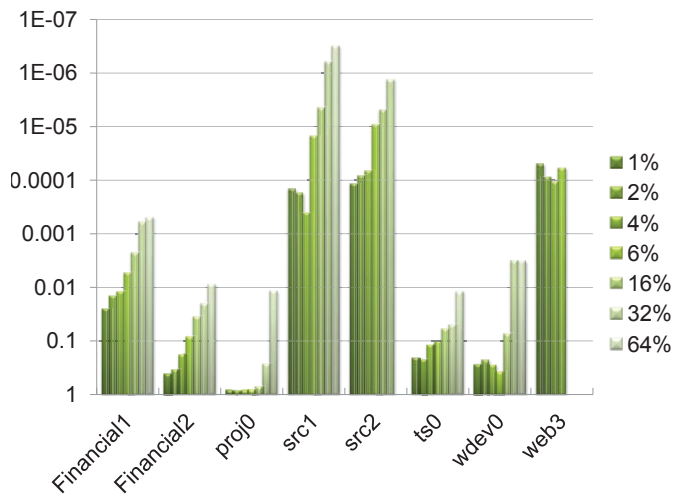


Figure 15: SE-UARC’s SSD allocation count at the highest sieve threshold normalized by that of baseline UARC. As the y-axis has been inverted, a higher bar represents a greater lifetime improvement due to fewer allocations

1) *Benefits of Sieving:* In [14], the authors show how sieving allocations improves the overall hit rate of an ensemble-level LRU cache (shared cache across all workloads) by 35% to 50%. Thus, an interesting question is whether sieving retains its edge in our multilevel scenario. Figure 14 shows the best execution time (across all thresholds) of SE-UARC normalized by baseline UARC’s execution time (no sieving). As can be seen, sieving offers marginal improvement at best under most traces. In the worst case, sieving deteriorates performance significantly at large cache sizes. This can be intuitively explained by the reduction in the number of SSD ARC hits.

Although not shown here due to lack of space, we also noticed that sieving failed to improve performance of even IARC. The best case reduction in IARC execution time was a meager 5% across all traces. Thus, unlike prior research, we believe that sieving cannot be used as a performance boosting technique in direct-attached multilevel caches, especially when used in collaboration with an adaptive caching algorithm like ARC.

2) *Ideal Lifetime Savings:* Although improving performance would have been an additional advantage of using sieving, saving flash lifetime is its primary objective. Figure 15 shows the number of allocations made by SE-UARC at the highest sieve threshold (64 accesses) normalized by the allocation count of baseline UARC for various configurations. It is important to note that the axis has been inverted and thus, a higher column implies a lower allocation count, which, in turn, translates to greater lifetime savings. Figure 16 shows the execution time of SE-UARC at the highest sieve threshold, normalized by baseline UARC execution time.

Clearly, sieving has a significant impact on flash lifetime as it reduces allocations in the SSD by several orders of magnitude under all traces. In addition, most traces show little to no performance degradation at low cache sizes despite such aggressive sieving. src1 and src2 are extremely cases

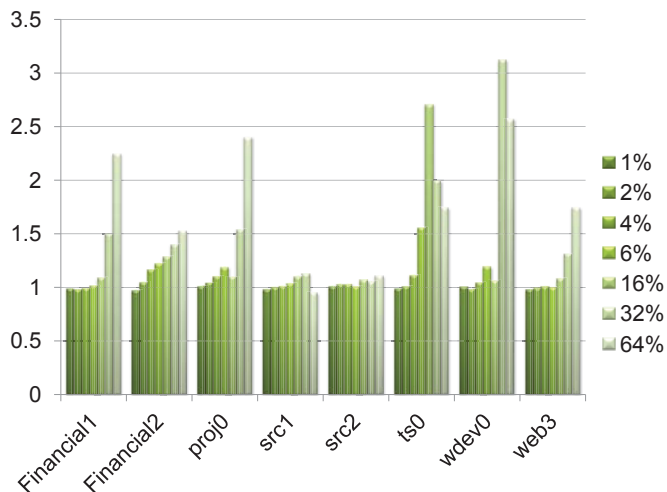


Figure 16: SE-UARC’s execution time at the highest sieve threshold normalized by baseline UARC’s execution time

with little degradation at all cache sizes. This illustrates how the performance impact of sieving is directly proportional to trace locality. Comparing these results with Figure 5, one can easily see that lower the locality of the trace (lower the cache effectiveness due to small cache size), higher the benefits of sieving.

B. Sieving And Cost-Awareness

In the presence of sieving, the total I/O cost of each block in RAM changes radically as a block can now be classified as either a “hot” block that passes the sieve test or a “cold” block that fails the test. Thus, the cost of evicting a hot block is different from that of evicting a cold block in both clean and dirty cases. Table VI summarizes the cost of various block types. Based on the table, we can classify blocks into five types, namely, cold-clean-write-next (no op), hot-clean and hot-dirty write-next (one SSD op), hot-clean and hot-dirty read-next (two SSD ops assuming data stays buffered in the SSD), cold-clean read-next and cold-dirty-write-next (one HDD op), and cold-dirty-read-next (two HDD ops). Thus, rather than extending CA-UARC with sieving (similar to the way we extended baseline UARC), we can use sieving to modify the replacement logic itself. To compare these two designs, we implemented sieving-enabled CA-UARC (SE-CA-UARC), and sieving-aware replacement in CA-UARC (SAR-CA-UARC).

To reiterate, while both SE-CA-UARC and CA-UARC are cost aware, SE-CA-UARC does not allocate blocks that fail the sieve test in the SSD ARC. In addition to this difference, SAR-CA-UARC also uses different cost lists to maintain the RAM cost segment when compared to CA-UARC (five as opposed to two). It is important to note here that sieving only alters RAM cost lists. Thus, both SE-CA-UARC and SAR-CA-UARC use three SSD cost lists.

On evaluating the two cache types, we found that the best-case execution time of SE-CA-UARC was identical to SAR-CA-UARC. This indicates that integrating sieving into the replacement logic produces little performance advantage.

Popularity	Status	NA	I/O Cost	List
Hot	Clean	R	1 SSD W + 1 HDD/SSD R	2
Hot	Clean	W	1 SSD W	1
Hot	Dirty	R	1 SSD W + 1 HDD/SSD R	2
Hot	Dirty	W	1 SSD W	1
Cold	Clean	R	1 HDD R	3
Cold	Clean	W	0	0
Cold	Dirty	R	1 HDD W + 1 HDD R	4
Cold	Dirty	W	1 HDD W	3

Table VI: Table lists the I/O costs of each block used to determine the RAM cost lists under SAR-CA-UARC. The list number identifies the order in which lists are chosen during demotion to free up RAM space. The SSD cost lists have not been listed as they are exactly the same as shown in Table IV.

However, on investigating this further, we found that while the relative improvement over baseline UARC was similar under both SE-CA-UARC and SAR-CA-UARC, the underlying source of improvement was different. As shown in Table VI, SAR-CA-UARC prioritizes eviction of “hot”-dirty blocks and “cold”-clean blocks over “cold”-dirty blocks as their I/O costs are different. In the absence of this logic, SE-CA-UARC, on the other hand, treats both “hot” and “cold”-dirty blocks similarly (Table IV). Thus, while SAR-CA-UARC optimized performance by reducing the number of “cold” flushes to the disk (at the expense of higher read misses), SE-CA-UARC optimized performance by reducing the number of read misses (at the expense of increased flushes).

While our cost model simulates a disk drive with balanced read/write latencies, there are several scenarios where this assumption is not true. For instance, in most enterprises, a persistent cache is typically used as a high-speed front end for a disk array. In such a scenario, it is well known that array writes might cost much more than reads depending on the redundancy technique used [6]. In such cases, SAR-CA-UARC would show a distinct performance advantage as it reduces the number of flushes. On the other hand, most caching implementations employ background destaging of dirty data to ensure free space availability in the cache and avoid blocking foreground operations. Combining destaging with a write-optimized layout for the disk tier would dramatically reduce the cost of flushing data. In such a scenario, SE-CA-UARC would definitely outperform SAR-CA-UARC.

C. SE-CA-UARC/IARC Comparison

Putting it all together, Figure 17 shows the best-case execution time of SE-CA-UARC, across all sieve thresholds and λ values, normalized by the best-case execution time of SE-IARC. We present only the results obtained while using SE-CA-UARC in combination with the ideal predictor here due to lack of space. Comparing this with Figure 11, one can see that with cost-awareness in place, SE-CA-UARC consistently outperforms IARC by a significant margin under most trace-cache size combinations.

Figure 18 depicts yet another benefit of exclusive caching by presenting SE-CA-UARC’s best-case SSD allocation normalized by that of SE-IARC. As can be seen, SE-CA-UARC also

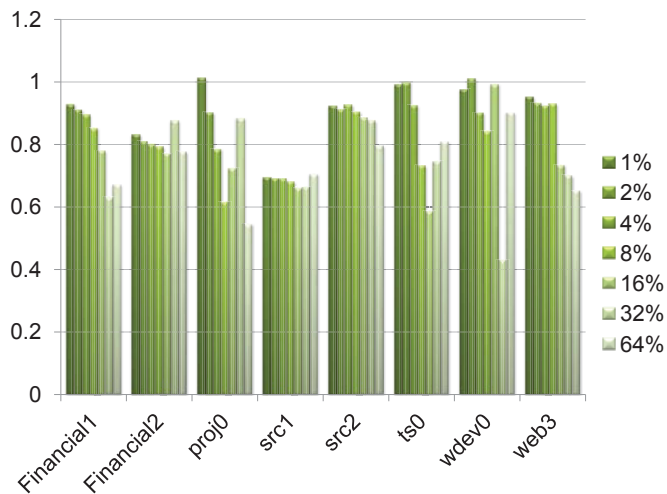


Figure 17: SE-CA-UARC’s best-case execution time (across all sieve thresholds) normalized by that of IARC

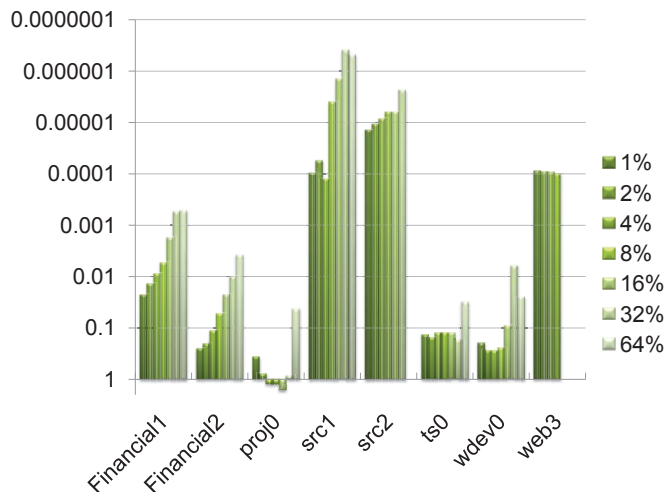


Figure 18: SE-CA-UARC’s SSD allocation count at the highest sieve threshold normalized by that of IARC

offers substantial reduction in the number of SSD allocations under almost all traces. Thus, exclusivity can be used not only to improve performance but also SSD lifetime.

VI. CONCLUSIONS

Despite being provisioned with large amounts of RAM and SSDs, several modern enterprise storage appliances manage the two cache levels in an inclusive fashion. By using a trace driven simulator, we showed how maintaining exclusivity in such direct-attached, multilevel caches not only improve performance but also improve SSD lifetime by reducing the number of allocations. Thus, we believe that exclusivity should be considered a first class citizen in designing next-generation storage systems. Although we showed the potential of cost-aware extensions in improving the performance of a read-write cache, more work is needed to bridge the gap between the accuracy of online predictors and an offline ideal predictor.

While we investigated the design alternatives involved in building a block-level exclusive cache, modern file systems

already manage a multilevel data cache, with RAM-resident buffer cache being the first level and flash-based SSDs being the second level. In addition, an exclusive file-level caching hierarchy could utilize readily-available semantic information to derive more accurate cost estimates, and hence, improve cache effectiveness. Thus, we believe that an interesting topic of future research is investigating exclusivity extensions to file systems.

ACKNOWLEDGMENT

This work has been supported by the European Research Council Advanced Grant 227874.

REFERENCES

- [1] “EMC Fast Cache - A Detailed Review,” EMC White Paper, 2011.
- [2] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey, “Awol: an adaptive write optimizations layer,” in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008.
- [3] Z. Chen, Y. Zhang, Y. Zhou, H. Scott, and B. Schiefer, “Empirical evaluation of multi-level buffer cache collaboration for storage systems,” *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, pp. 145–156, Jun. 2005.
- [4] Z. Chen, Y. Zhou, and K. Li, “Eviction-based cache placement for storage caches,” in *Proc. of the USENIX Ann. Tech. Conf.*, 2002.
- [5] B. S. Gill, “On multi-level exclusive caching: offline optimality and why promotions are better than demotions,” in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008.
- [6] B. S. Gill and D. S. Modha, “Wow: wise ordering for writes - combining spatial and temporal locality in non-volatile caches,” in *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, 2005.
- [7] S. Jiang and X. Zhang, “Ulc: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches,” in *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS’04)*, 2004.
- [8] K. H. Lee, I. H. Doh, J. Choi, D. Lee, and S. H. Noh, “Write-aware buffer cache management scheme for nonvolatile ram,” in *Proceedings of the third conference on IASTED International Conference: Advances in Computer Science and Technology*, 2007.
- [9] X. Li, “Second-tier cache management to support dbms workloads,” Ph.D. dissertation, University of Waterloo, 2011.
- [10] N. Megiddo and D. S. Modha, “Arc: A self-tuning, low overhead replacement cache,” in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.
- [11] D. Narayanan, A. Donnelly, and A. Rowstron, “Write off-loading: Practical power management for enterprise storage,” *Trans. Storage*, vol. 4, no. 3, pp. 10:1–10:23, Nov. 2008.
- [12] Y. Oh, J. Choi, D. Lee, and S. H. Noh, “Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems,” in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, 2012.
- [13] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, “The case for ramcloud,” *Commun. ACM*, vol. 54, no. 7, pp. 121–130, Jul. 2011.
- [14] T. Pritchett and M. Thottethodi, “Sievestore: a highly-selective, ensemble-level disk cache for cost-performance,” in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010.
- [15] U. T. Repository, “<http://traces.cs.umass.edu/index.php/storage/storage>.”
- [16] T. M. Wong and J. Wilkes, “My cache or yours? making storage more exclusive,” in *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, 2002.
- [17] G. Yadgar, M. Factor, K. Li, and A. Schuster, “Management of multi-level, multiclient cache hierarchies with application hints,” *ACM Trans. Comput. Syst.*, vol. 29, no. 2, pp. 5:1–5:51, May 2011.
- [18] G. Yadgar, M. Factor, and A. Schuster, “Karma: know-it-all replacement for a multilevel cache,” in *Proceedings of the 5th USENIX conference on File and Storage Technologies*, 2007.